**Java线程：并发协作-生产者消费者模型**

**对于多线程程序来说，不管任何编程语言，生产者和消费者模型都是最经典的。就像学习每一门编程语言一样，Hello World！都是最经典的例子。**

**实际上，准确说应该是“生产者-消费者-仓储”模型，离开了仓储，生产者消费者模型就显得没有说服力了。**

**对于此模型，应该明确一下几点：**

**1、生产者仅仅在仓储未满时候生产，仓满则停止生产。**

**2、消费者仅仅在仓储有产品时候才能消费，仓空则等待。**

**3、当消费者发现仓储没产品可消费时候会通知生产者生产。**

**4、生产者在生产出可消费产品时候，应该通知等待的消费者去消费。**

**此模型将要结合java.lang.Object的wait与notify、notifyAll方法来实现以上的需求。这是非常重要的。**

**/\*\*   
\* Java线程：并发协作-生产者消费者模型  
\*   
\* @author leizhimin 2009-11-4 14:54:36**

**生产者和消费者同步代码块的锁是同一个对象  
\*/ public class Test {  
        publicstaticvoid main(String[] args) {  
                Godown godown = new Godown(30);  
                Consumer c1 = new Consumer(50, godown);  
                Consumer c2 = new Consumer(20, godown);  
                Consumer c3 = new Consumer(30, godown);**

**Producer p1 = new Producer(10, godown);  
                Producer p2 = new Producer(10, godown);  
                Producer p3 = new Producer(10, godown);  
                Producer p4 = new Producer(10, godown);  
                Producer p5 = new Producer(10, godown);  
                Producer p6 = new Producer(10, godown);  
                Producer p7 = new Producer(80, godown);  
  
                c1.start();   
                c2.start();   
                c3.start();   
                p1.start();   
                p2.start();   
                p3.start();   
                p4.start();   
                p5.start();   
                p6.start();   
                p7.start();   
        }   
}   
  
/\*\*   
\* 仓库   
\*/   
class Godown {**

**public static final int max\_size = 100;//最大库存量**

**public static int curnum=0;    //当前库存量**

**Godown() {**

**}**

**Godown(int curnum) {**

**this.curnum = curnum;**

**}**

**/\*\***

**\* 生产指定数量的产品**

**\* @param neednum**

**\*/**

**public synchronized void produce(int neednum) {**

**//测试是否需要生产**

**while (neednum + curnum > max\_size) {**

**System.out.println("要生产的产品数量" + neednum +"超过剩余库存量" + (max\_size - curnum) +"，暂时不能执行生产任务!");**

**try {**

**//当前的生产线程等待**

**wait();**

**} catch (InterruptedException e) {**

**e.printStackTrace();**

**}**

**}**

**//满足生产条件，则进行生产，这里简单的更改当前库存量**

**curnum += neednum;**

**System.out.println("已经生产了" + neednum +"个产品，现仓储量为" + curnum);**

**//唤醒在此对象监视器上等待的所有线程**

**notifyAll();**

**}**

**/\*\***

**\* 消费指定数量的产品**

**\* @param neednum**

**\*/**

**public synchronized void consume(int  neednum) {**

**//测试是否可消费**

**while (curnum < neednum) {**

**try {**

**//当前的生产线程等待**

**wait();**

**} catch (InterruptedException e) {**

**e.printStackTrace();**

**}**

**}**

**//满足消费条件，则进行消费，这里简单的更改当前库存量**

**curnum -= neednum;**

**System.out.println("已经消费了" + neednum +"个产品，现仓储量为" + curnum);**

**//唤醒在此对象监视器上等待的所有线程**

**notifyAll();**

**}**

**}   
/\*\*   
\* 生产者   
\*/   
class Producer extends Thread {   
        private int  neednum;                //生产产品的数量  
        private Godown godown;            //仓库  
  
        Producer(int neednum, Godown godown) {  
                this.neednum = neednum;  
                this.godown = godown;  
        }   
  
        public void run() {  
                //生产指定数量的产品  
                godown.produce(neednum);   
        }   
}   
  
/\*\*   
\* 消费者   
\*/   
class Consumer extends Thread {   
        private int  neednum;                //消费产品的数量  
        private  Godown godown;            //仓库  
  
        Consumer(int neednum, Godown godown) {  
                this.neednum = neednum;  
                this.godown = godown;  
        }   
  
        publicvoid run() {  
                //消费指定数量的产品  
                godown.consume(neednum);   
        }   
}**

**已经生产了10个产品，现仓储量为40  
已经生产了10个产品，现仓储量为50  
已经消费了50个产品，现仓储量为0  
已经生产了80个产品，现仓储量为80  
已经消费了30个产品，现仓储量为50  
已经生产了10个产品，现仓储量为60  
已经消费了20个产品，现仓储量为40  
已经生产了10个产品，现仓储量为50  
已经生产了10个产品，现仓储量为60  
已经生产了10个产品，现仓储量为70  
  
Process finished with exit code 0**

**说明：**

**对于本例，要说明的是当发现不能满足生产或者消费条件的时候，调用对象的wait方法，wait方法的作用是释放当前线程的所获得的锁，并调用对象的notifyAll()方法，通知（唤醒）该对象上其他等待线程，使得其继续执行。这样，整个生产者、消费者线程得以正确的协作执行。**

**notifyAll() 方法，起到的是一个通知作用，不释放锁，也不获取锁。只是告诉该对象上等待的线程“可以竞争执行了，都醒来去执行吧”。**

**本例仅仅是生产者消费者模型中最简单的一种表示，本例中，如果消费者消费的仓储量达不到满足，而又没有生产者，则程序会一直处于等待状态，这当然是不对的。实际上可以将此例进行修改，修改为，根据消费驱动生产，同时生产兼顾仓库，如果仓不满就生产，并对每次最大消费量做个限制，这样就不存在此问题了，当然这样的例子更复杂，更难以说明这样一个简单模型。**

IMG_256

**IMG_256IMG_256IMG_256IMG_256IMG_256/\*\*生产者消费者问题，涉及到几个类**

**\* 第一，这个问题本身就是一个类，即主类**

**\* 第二，既然是生产者、消费者，那么生产者类和消费者类就是必须的**

**\* 第三，生产什么，消费什么，所以物品类是必须的，这里是馒头类**

**\* 第四，既然是线程，那么就不是一对一的，也就是说不是生产一个消费一个，既然这样，多生产的往哪里放，**

**\* 现实中就是筐了，在计算机中也就是数据结构，筐在数据结构中最形象的就是栈了，因此还要一个栈类**

**\*/**

**package thread;**

**public class ProduceConsume {**

**public static void main(String[] args) {**

**SyncStack ss = new SyncStack();//建造一个装馒头的框**

**Producer p = new Producer(ss);//新建一个生产者，使之持有框**

**Consume c = new Consume(ss);//新建一个消费者，使之持有同一个框**

**Thread tp = new Thread(p);//新建一个生产者线程**

**Thread tc = new Thread(c);//新建一个消费者线程**

**tp.start();//启动生产者线程**

**tc.start();//启动消费者线程**

**}**

**}**

**//馒头类**

**class SteamBread{**

**int id;//馒头编号**

**SteamBread(int id){**

**this.id = id;**

**}**

**public String toString(){**

**return "steamBread:"+id;**

**}**

**}**

**//装馒头的框，栈结构**

**class SyncStack{**

**int index = 0;**

**public static final int Max\_Size = 6;//最大库存量**

**SteamBread[] stb = new SteamBread[Max\_Size];//构造馒头数组，相当于馒头筐，容量是6**

**//放入框中，相当于入栈**

**public synchronized void push(SteamBread sb){**

**while(index==stb.length){//筐满了，即栈满，**

**try {**

**this.wait();//让当前线程等待**

**} catch (InterruptedException e) {**

**// TODO Auto-generated catch block**

**e.printStackTrace();**

**}**

**}**

**this.notifyAll();//唤醒在此对象监视器上等待的所有线程，即消费者线程**

**stb[index] = sb;**

**this.index++;**

**}**

**//从框中拿出，相当于出栈**

**public synchronized SteamBread pop(){**

**while(index==0){//筐空了，即栈空**

**try {**

**this.wait();**

**} catch (InterruptedException e) {**

**// TODO Auto-generated catch block**

**e.printStackTrace();**

**}**

**}**

**this.notifyAll();**

**this.index--;//push第n个之后，this.index++，使栈顶为n+1，故return之前要减一**

**return stb[index];**

**}**

**}**

**//生产者类，实现了Runnable接口，以便于构造生产者线程**

**class Producer implements Runnable{**

**SyncStack ss = null;**

**Producer(SyncStack ss){**

**this.ss = ss;**

**}**

**@Override**

**public void run() {**

**// 开始生产馒头**

**for(int i=0;i<20;i++){**

**SteamBread stb = new SteamBread(i);**

**ss.push(stb);//同步方法，生产成功则执行下面代码**

**System.out.println("生产了"+stb);**

**try {**

**Thread.sleep(10);//每生产一个馒头，睡觉10毫秒**

**} catch (InterruptedException e) {**

**// TODO Auto-generated catch block**

**e.printStackTrace();**

**}**

**}**

**}**

**}**

**//消费者类，实现了Runnable接口，以便于构造消费者线程**

**class Consume implements Runnable{**

**SyncStack ss = null;**

**public Consume(SyncStack ss) {**

**super();**

**this.ss = ss;**

**}**

**@Override**

**public void run() {**

**// TODO Auto-generated method stub**

**for(int i=0;i<20;i++){//开始消费馒头**

**SteamBread stb = ss.pop();**

**System.out.println("消费了"+stb);**

**try {**

**Thread.sleep(100);//每消费一个馒头，睡觉100毫秒。即生产多个，消费一个**

**} catch (InterruptedException e) {**

**// TODO Auto-generated catch block**

**e.printStackTrace();**

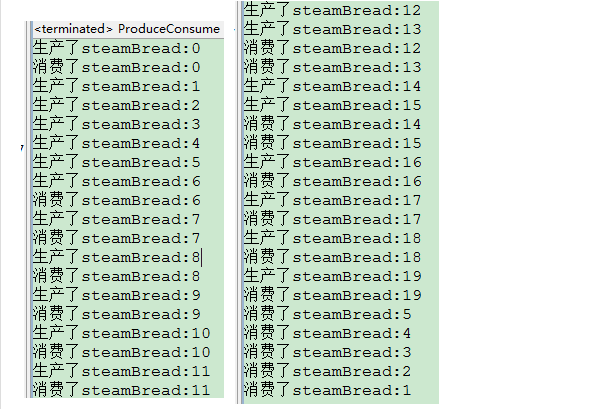
**}**

**}**

**}**

**}**

**运行结果：**

****